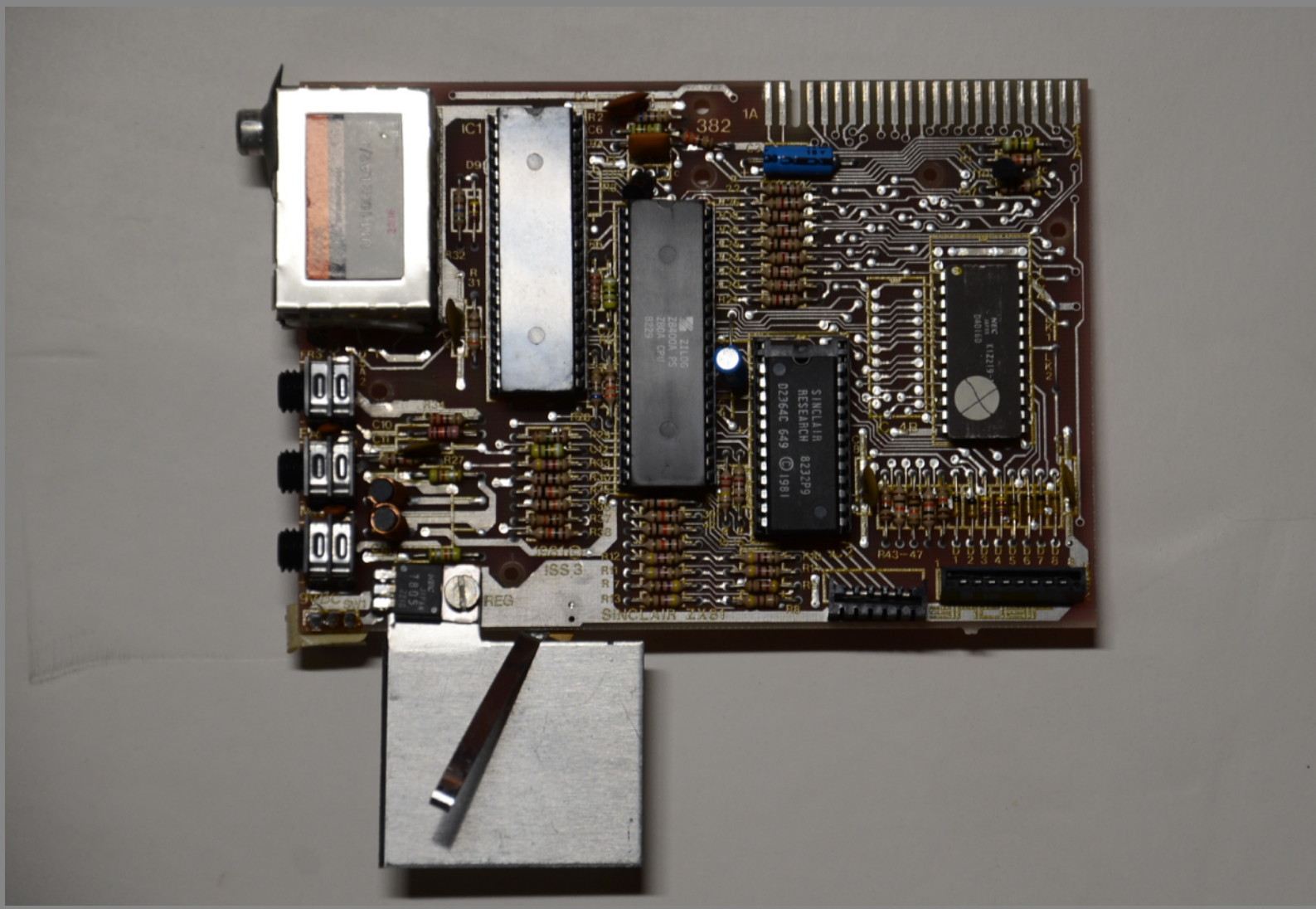
2x2ine

Issue #10 **January 2024**



ZXzine

Table of Contents

Published by:
Timothy Swenson
swenson_t@sbcglobal.net
swensont@lanset.com

ZXzine is published as a service to the Timex/Sinclair community. Writers are invited to submit articles for publication. Readers are invited to submit article ideas.

Created using Open Source Tools:

- OpenOffice
- Scribus
- Gimp
- SZ81
- Zesarux

Editorial

I've written enough articles for another issue of ZXzine. Hopefully readers will find them interesting.

At the Timex/Sinclair User Group zoom meetings there has been discussion of a printed magazine with ZXzine suggested as the source. I am hoping to get others to write for this print version to make it more encompassing of the T/S scene.

It is unclear if a set number of copies will be printed or if we will go the print-on-demand route. This is still something that is to be explored.

If you have an article idea or you have some articles that you would like to see, please let me know. There is no set date on when the print version will appear, or exactly how it will appear, but the first step is to get the articles written.

TimeMachine on Zesarux

I thought I would take another crack at TimeMachine, the Basic compiler for the T/S 2068 and with some pointers from David Anderson, I was able to get it working. Here is how I got it working on Zesarux.

TimeMachine is available from David's Website (www.timexsinclair.com). TimeMachine is a T/S 2068 port of Hisoft BASIC for the ZX Spectrum. The Hisoft BASIC manual can be found online via a Google Search.

First, there are some changes that need to be made to Zesarux. When loading a tape, the default behavior is to do a reset and then load the program. With TimeMachine loaded above RAMTAP, the reset wiped it out.

In the Zesarux menu, go to Settings -> Storage -> Autoload Medium. Once this is set, when you load a tape, you have to use the LOAD command to load the program. Normally I like the autoload feature, but it won't work with TimeMachine, so I know to set it when running TimeMachine.

When saving with Zesarux, the Output tape must be set. In the menu, go to Storage -> Tape and select Output Tape. There you can browse to find the right file, or you can hit Tab twice and then enter the name of a new file.

With the issues of Zesarux out of the way, on to TimeMachine.

Load TimeMachine through the normal process. Once it loads and the initial screen is drawn, type R to run the program. The screen will clear and TimeMachine is ready. Use the Zesarux menu to select the input file with the program to compile. Then type LOAD "" to load the program.

TimeMachine needs two compiler directives to know where to compile. The directives are stored in a REM statement. The first directive is to tell the compiler where to start compiling. The Hisoft manual has the compiler directive as:

```
10 REM : OPEN #
```

But TimeMachine needs a "!" instead of a ":".

```
10 REM ! OPEN #
```

The "OPEN #" is the keyword, not just the text. At the end of your program is another compiler directive, telling the compiler where to stop:

```
XXX REM ! CLOSE #
```

I created my program with zmakebas, and it did not tokenize any of the lines in the REM statements, so I had to manually enter the compiler directives once the program was loaded.

The next step is to compile. Just hit the * key and then C and the compiler will start. The screen will look odd and stop. Just hit space to have the compiler move on. Another odd screen and hit the space bar again and you are done. There will be text on the screen about the compilation, how many bytes is the compiled program and the commands to SAVE and LOAD the program.

With my program, it was stored at 60831 and was 2915 bytes. To save to a TAP file, I created a new

output file and then typed:

```
SAVE "XXXX"CODE 60831,2915
```

Once saved, I reset the emulator and then loaded my program with the command:

```
LOAD "XXXX"CODE 60831
```

And then to run:

```
RANDOMIZE USR 60831
```

My program originally would not compile, as I was using the VAL statement which TimeMachine does not support. I changed my code to not use VAL and it then compiled.

After I got that all working, my next step was how to create a loader for the compiled program. The virtual tape drive on Zesarux is a streaming device and not random access. If there are three files on the tape and you load file number 3, you have to "rewind" the tape to then load file number 1.

The question is how to save a loader program after the compiler has run, but before saving the compiled program. Luckily, TimeMachine as the "*ERASE" command (where ERASE is the keyword). It deletes the BASIC program and variables in memory, but does not touch the compiled program.

Once the program is compiled, I then created a new TAP file using the Storage -> Tape -> Output option. I make sure to write down the SAVE options that TimeMachine showed on the screen after compilation completed. I then did the *ERASE command. Now I can type a short loader program. In my example, the BASIC program was CTEST1. The BASIC program was in "ctest1.tap". The new output file is "ctest1c.tap", where C stands for compiled. The loader program is this:

```
10 LOAD "CTEST1C" CODE 65066
20 RANDOMIZE USR 65066
```

I then save the loader program as "CTEST1L" to virtual tape. I then type the command:

```
SAVE "CTEST1C" CODE 65066,272
```

Now I have in the file "ctest1c.tap" the virtual tape files, CTEST1L and then CTEST1C. I then deselect "ctest1c.tap" as the output file and select it as the input file. Next I LOAD "" then type RUN. It loads the compiled program and runs it.

Now I can distribute the "ctest1c.tap" file to anyone and they don't need to know anything about TimeMachine or that the program is compiled. They just load the first program on the tape and away it goes.

Assembly Parameter Passing

I am writing a assembly language routine that needs to send data back to a BASIC program and to get data from a BASIC program. Getting data from the routine is fairly simple if it a single value. Getting data to the routine is more complex as the USR call does not allow for arguments. This article explores two ways to get data back and forth from assembly to BASIC.

The Assembly Language Routine

Without going into too much detail, the routine needed to send 2 numbers back to BASIC and then have BASIC send 2 numbers to the routine. The numbers are the line and column for a screen location. The process I used to get the data into and out of the assembly routine is dependent on just needing two numbers and esp. screen location numbers. If I needed different data, I might go a different route.

Getting Data to BASIC

When making a USR call like this:

```
LET A = USR 16514
```

The contents of the BC register pair is loaded into the variable A. My need was two numbers that I knew would be only 8-bit, I could use the B register for one number and the C register for the other. I

just had to take the 16-bit number returned by the routine and break it into two 8-bit numbers.

The test assembly routine was this:

```
LD B,15
LD C,25
RET
```

The two values were loaded into the appropriate registers and then the routine returned to BASIC.

The BASIC program to get the data was this:

```
1 REM #######
2 LET A = USR 16514
3 LET B = INT (A/256)
4 LET C = A-(256*B)
```

The REM statement was there as a place to put the assembly language routine. Lines 3 and 4 break down the 16-bit number into the original two 8-bit numbers.

The assembly language routine is compiled with Pasmo into a .bin file and not a .P file. The BASIC program is converted into a .P file with zxtext2p. I then use dd to write the .bin file into the area of the .P file where the REM statement is.

The .P file is then loaded into an emulator, ran and the program shows the two values listed in the assembly program. In the zip file this is passtest1.*

Getting Data from BASIC

Looking through the ZX81 system variables, I found S-POSN. It stored the column and line numbers from the last print statement. To get an exact pair of numbers, PRINT AT can be used to get the data needed into S-POSN. The BASIC program was this:

```
1 REM ##.......
2 PRINT AT 10,15;
3 LET A = USR 16514
```

Normally the "LET A = USR" line is used when you want to get data back. Other way of calling the routine is "PRINT USR ....", but the return value from the routine will be printed on the screen. That is not what I wanted.

The assembly language part was a little more complicated:

```
PRINTEQU  $0010
PRINTAT   EQU  $08f5

      LD   A,($4039)
      LD   C,A
      LD   A,33
      SUB  C
      LD   C,A
      LD   A,($403A)
      LD   B,A
      LD   A,24
      SUB  B
      LD   B,A
      CALL PRINTAT
      LD   A,$08
      CALL PRINT
      RET
```

In playing with S-POSN, I found that the value is not really the value set by PRINT AT. When I did PRINT AT 0,0, the value in S-POSN was 33 and 24. As I incremented the numbers for the PRINT AT, the values in S-POSN decreased. To get the real value that has been set by PRINT AT, I had to subtract 33 and 24 from the S-POSN values.

With Z80 assembly language, subtraction is only allowed with the A register. There is also a limitation on which registers can have data from memory copied to them. This meant that I had to get the S-POSN values into A, then copy to another register. Then load A with either 33 or 24 and then subtract B or C from A to get the value that I needed.

The routine then just prints a grey character on the screen at the location of values sent from BASIC. In the zip file this is passtest2.*

An Alternative Process

Another way to do this is put the values needed into specific memory locations. BASIC can PEEK to get the values and POKE to pass the values. I've

been using a REM statement as a safe place for the assembly routine, it is also a safe place to store the values. The simplest location is the very beginning of the REM statement, which shifts the start of the assembly routine by two bytes.

Passing from Assembly to BASIC

The assembly routine just loads values into the A register and then moves them into the memory locations:

```
DEFB $00
DEFB $00

    LD   A,15
    LD   (16514),A
    LD   A,25
    LD   (16515),A
    RET
```

The BASIC program calls the assembly routine and then PEEKs the values:

```
1 REM 12345678901234
2 let a = USR 16516
3 let a = peek(16514)
4 let b = peek(16515)
5 print a
6 print b
```

In the zip files, this is passtest3.*

Passing from BASIC to Assembly

To get the values from BASIC to assembly, the values POKEd into the memory locations:

```
1 REM 12345678901234567890
2 poke 16514,15
3 poke 16515,25
4 let a = USR 16516
```

Once the values are stored, the assembly routine is called, which takes the values and print a grey character at the PRINT AT location:

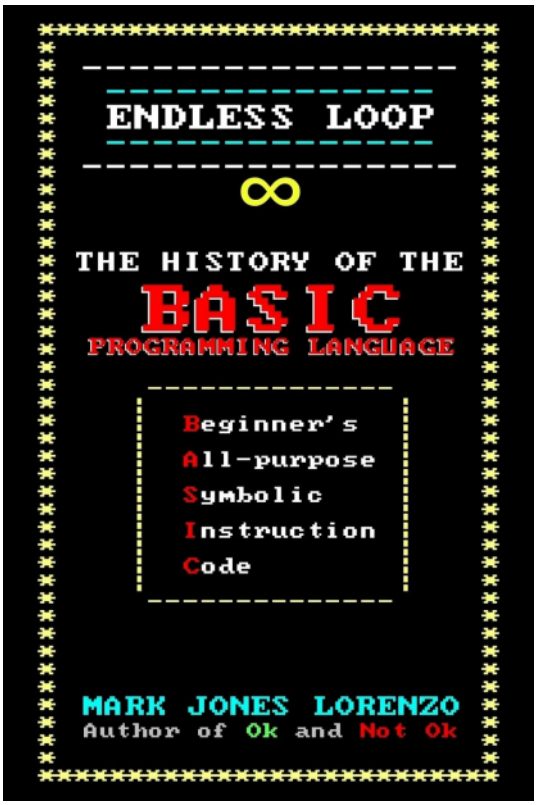
```
DEFB $00
DEFB $00

    LD   A,(16514)
    LD   B,A
    LD   A,(16515)
    LD   C,A
    CALL PRINTAT
    LD   A,$08
    CALL PRINT
    RET
```

In the zip file this is passtest4.*

Two Books on BASIC

In the last year I picked up two books by Mark Jones Lorenzo, "Endless Loop: The History of the BASIC Programming Language" and "Gosub Without Return: Between the Lines of the BASIC Programming Language."

The first book is a straight forward book documenting the history of BASIC, from its inception at Dartmouth and the design goal of it being used by non-computer students, to the end days of BASIC in the 90's.



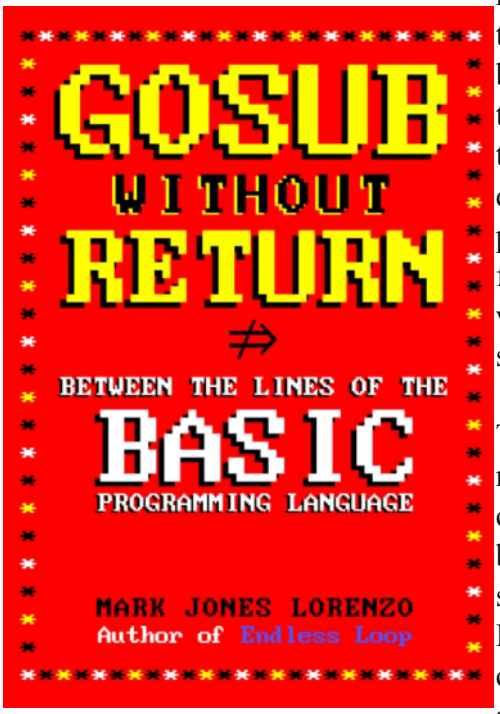
The book also talks about the many versions of BASIC and how a lot varied from standard BASIC, mostly because of the limited memory of early computer systems. The designers of BASIC did not copyright the language as they wanted it to spread and be a benefit of all. This helped lead to the many dialects of BASIC.

The book focuses on the popular BASICs of the time, Micrsoft BASIC, GW-BASIC and later QBasic. The early microcomputers had different BASICs from many authors, but as larger computer companies came onto the scene, quite a number of them contracted with MicroSoft for their version of BASIC. As awful as MicroSoft BASIC was, it dominated the market.

There is a page or two that talks about BASIC for the BBC Micro and SuperBASIC for the QL, but nothing really of Sinclair BASIC.

For someone into retro-computers, I found the book an enjoyable read. There was a lot I did not know about the early days of BASIC. The books is well researched and well written. The book is a paperback and appears to be published via print on demand. The book came out in 2017, but my book was printed in late 2022. I like the physcial size of the book. Small enough the comfortably hold in my hands to read.

The second book "Gosub without Return" is larger than the first in both the number of pages and the format size. This book is harder to classify than the first. I get the impression that as the author was researching the first book, all of the material that did not quite fit the premis of the first book went into the second book.



The chapters not are well connected but they are still about BASIC. One chapter is about BASIC standardization, another on the origins of TI BASIC, another on Sister Mary Keller, who some say helped create BASIC, etc. There are two chapters on "The UK Connection" that has more details on British computers. The second chapter of the UK connection is on Usborne books. There are so many references to other books that you could spend weeks tracking them down and reading them. Like the first book, this one is very well researched.

Since the book is not a history that is trying to tell a complete story, it is easy to just skip a chapter if it does not interest you. The first book was more interesting, but tthe second one as still a good read.

The books are cheap, with Amazon listing the first book for $10 and the second one for $14. I found them both enjoyable and to recommend them to anyone still proramming in BASIC.

Lunar Lander

Recently the book "50 years of Text Games" by Aaron Reed, was released via Kickstarter and I was able to get a copy. The book starts off with some of early and better known text-based computer games (at least for someone that got into computers in the early 80's).

The second game documented is ROCKET, the original name for a lunar lander game. It was written in late 1969 and published in January 1970. After watching the Apollo 11 lunar landing, Jim Storer, a high school student, thought that it could be turned into a computer game.

With access to a PDP-8, Storer wrote the program in the FOCAL language. FOCAL stands for “Formulating On-line Calculations in Algebraic Language” as was an interpretive language where the keywords were reduced to a single letter, since the PDP-8 was a small memory machine. In FOCAL here is the core part of the program:

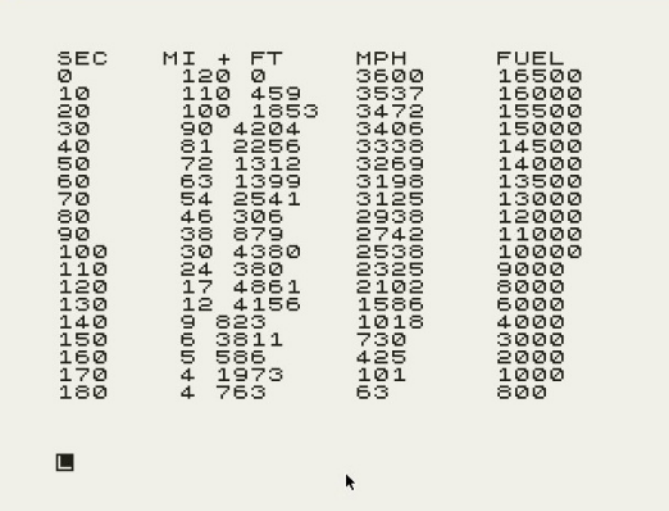
```
03.10 I (M-N-.001)4.1; I (T-
.001)2.1;S S=T
03.40 I ((N+S*K)-M)3.5,3.5; S
S=(M-N)/K
03.50 D 9; I (I)7.1,7.1; I
(V)3.8,3.8;I (J)8.1
3.80 D 6; G 3.1
```

Line 03.10 starts with I which means IF, so in

BASIC it would mean:

```
IF (M-N-.001) < 0 THEN GOTO 4.1
```

The game was published in the DECUS Program Library. Later David Ahl converted the program to BASIC about 1971 and published it in EDU, the DEC newsletter. It was later published in "101 BASIC Computer Games" in 1973 as ROCKET and as Lunar LEM Rocket in "BASIC Computer Games" in 1973 and in the more known 1978 edition.



I thought it might be a good idea to get the game running on the ZX81. I know there are a few Lunar Lander games for the ZX81, but all of them a graphical. I wanted to hark back to the early version.

"Basic Computer Games" is a book that I used to have and is available from the Internet Archive. The downloaded version of the book does not have the best quality graphics, so I had to view the program directly on the Internet Archive. With the lower-res scan of the PDF it was hard to say if a character was a "*" or a "+".

The program required some modification as the ZX81 does not support multiple statements on a line. I had to adjust the display to fit the ZX81 and add a pause statement to keep the screen from needing a "CONTINUE" after it filled up from the opening of the game.

I ran the program using the example given in the book and I got the same results, so I think it is working properly.

After the initial instructions screen, four items are printed on the screen; Seconds, Elevation of the lander (in miles and feet), the speed of the lander and the amount of fuel left. After this is printed, the cursor will appear at the bottom of the screen ready for numeric input. Enter the amount of fuel that you want to burn for the next ten seconds. Once that calculation is done, another set of numbers is shown and the computer is ready for another amount of fuel to burn. The input can be from 8 to 200, with 200 having the rocket motors at full blast.

There has to be a balance of fuel and descent speed. If the lander comes in too slow, it can run out of fuel. Wait too long to kick in the engine and it can be too late to slow down the lander.

Comparing ZXbasic and TimeMachine

There are two available BASIC compilers for the T/S 2068, TimeMachine and ZXbasic. TimeMachine is native on the T/S 2068 and ZXbasic is a cross compiler, written in Python, that runs on Windows, Linux and MacOS. This article will do a short comparision between them.

ZX BASIC Language Support

Both compilers do not compile the full syntax of ZX Basic.

TimeMachine is based on Hisoft BASIC for the ZX Spectrum. The documentation for Hisoft BASIC is mostly accurate for TimeMachine. There is a secion of the Hisoft BASIC user guide that details the the limitations of the compiler. It does not allow for VAL A$, but VAL "12345" is allowed. Arrays of no more than 3 dimensions is permitted. BREAK key is disabled. A number of immediate commands (CLEAR, CONTINUE, ERASE, FORMAT, etc) can not be compiled.

ZXBasic compiles most Sinclair BASIC, but some commands are totally different. All input is a string. VAL must be used to convert to a number:

```
u$ = input(10)
year = val(u$)
```

input(10) means to get no more than 10 characters

of input.

ZXBasic is also based on FreeBASIC, so it can handle a more structured version of BASIC. IF statements can have multiple lines and allows for an ELSE statement. Besides FOR..NEXT it also supports WHILE..END WHILE and DO...LOOP. Line numbers are not needed, so it uses labels for GOTO and GOSUB statements.

Steps to Compile

TimeMachine takes a number of steps to compile a program. The program is first written with a text editor, then converted to a TAP file with zmakebas. The program is then loaded into an emulator where TimeMachine was first loaded. The program is compiled, and then saved out to another .TAP file.

ZXbasic takes the program in the text file and compiles it to a .TAP file that is ready to run. Overall the process with ZXBasic is simpler and faster.

Speed

I did a couple of speed comparisons with the two compilers. The first program is a simple print to the screen test:

```
10 CLS
20 FOR X=1 TO 31
30 FOR Y = 1 TO 21
40 PRINT ".";
50 NEXT Y
60 NEXT X
```

For TimeMachine additional lines where added for the compiler directives. Nothing was needed for ZXbasic.

Once compiled, I ran both programs. For TimeMachine it took about 2-3 seconds for the screen to fill up. It was slow enough that you could see the progress. ZXbasic was about 1 second. It was too fast to see the progress. Running the program in BASIC took about 6 seconds.

The second program that I tested with is my cellular automata program that generates some nice graphics. In BASIC it is achingly slow. With TimeMachine it was faster, but still fairly slow. With ZXbasic, it was not quick, but it was the fastest of the three programs and was not too slow to watch.

Comparing

If I had to choose between the two compilers, I would go with ZXbasic. It produces faster code and is only a single step to get the end result.

I did find a bug in ZXbasic with printing a floating point number. Once it prints the number the screen gets funny and the program hangs. I tested this out on the original Spectrum version of ZXbasic and the issue is still seen there.

For some programs that I was going to do, not printing out a floating point number is a show stopper, but it should be fine for other programming.

There are a number of ZXBasic programs written for the Spectrum that could be ported to the T/S 2068 if the source code is available.

ZX81 "Mouse"

Something recently got me thinking about having a mouse on a ZX81. At first I was thinking of a traditional mouse with a movable arrow. This might be possible with hi-res graphics, but I've never tried using hi-res graphics.

I though through about how a mouse works using DOS, which uses a text-based cursor which moves about the screen like a mouse. The cursor is just a text character. Something like this could work on the ZX81.

I had been working on assembly program that took keyboard input and moved a character around the screen. I was thinking I might eventually create a game, but the mouse idea was perfect for the assembly that I had already done.

Initially I was thinking about making the cursor a specific character and I would have to save the existing character that was on the screen before printing the cursor. Then I had the idea of inverting

the character on the screen. Adding 128 to any character will create the inverse character. Adding another 128 will get you back to the original character, because the 8-bit register will "roll" over and the carry bit can be ignored.

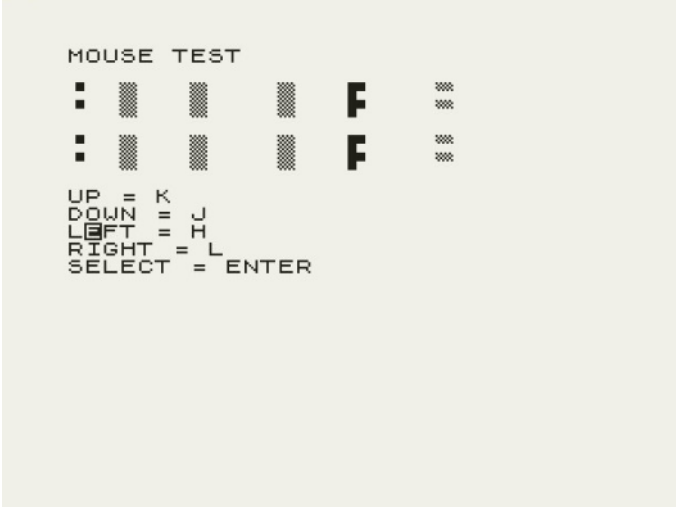
I created a proof of concept program in pure assembly. The program drew some characters on the screen, drew the cursor and then the user could move the cursor around the screen.

Originally I had the program use the arrow keys, but that was not too convenient. I then switched to the Unix "arrow" keys, the same ones used with VI or "less":

h - left
j - down
k - up
l - right

These were keys that I have been using for years, so I went with them. The mouse "button" to select was originally "0" but I changed it to "enter". If you are using your right hand to move the cursor it is pretty easy to hit the enter key.

Once the proof-of-concept was completed, the next step was to create an assembly routine that could be called from BASIC. The routine would move the cursor around the screen and when "enter" was hit,



it would return to BASIC passing the last location of the cursor. Then the BASIC program could take some action based upon where the cursor was.

The routine is in mouse.asm. A simple BASIC program is mousetest.bas. The assembly program was compiled into a .bin file with PASMO. Once I knew the size of the .bin file, I put a REM statement in mousetest.bas that is large enough to hold the .bin file. I "compiled" mousetest.bas with zxtext2p. I merged the .bin file with the .P file with the tool "dd", like this:

```
dd conv=notrunc seek=121 bs=1
if=mouse.bin of=mousetest.p
```

Initially the program did not work, but when I reached out to the Timex/Sinclar mailing list, Ryan Gray pointed out that I was missing an ORG statement.

There is a delay built into the routine. Without it the cursor is way too fast. Depending on the platform being used, the delay might need to be changed to be longer or shorter.

Mandlebrodt on the T/S 2068

[David Anderson, who runs www.timexsinclair.com, recieved a box of material from SyncWare News for archiving. In the box was a number of submitted articles that did not get published. David has sent me this article and program from Gerald W. Goegelein, originally submitted in Feb. 1987. David also ran the program, provided screenshots and a .tzx file with the program and data file, where you can GO TO 1000 to get directly to the menu and then load the data file. - Ed]

In looking for something to do I found myself reading back issues of SyncWare News for articles passed over the first time around. The Mandelbrot Plot program in Volume 3 Number 5 caught my eye. Those who read the article by Mr. Nachbaur will remember that the Mandelbrot program creates "computer art", mathematically created patterns otherwise known as fractals. These are in endless detail, many are very beautiful and many have smaller and smaller detail within themselves to explore. This looked interesting, so I entered the original listing to see how it would work.

After running it on my ZX-81 it became apparent that: 1) the screen display obtained using the "PREVIEW" option bore little resemblance to the printout, and 2) the printout consumed a lot of

expensive paper (not to mention the amount of time to do the print!). These discoveries prompted me to convert the program to run on the 2068 where it would be possible to properly display the picture without printing it.

As the author of the program (Mr. Nachbaur) noted, the 2068 does not have enough RAM available to support the 192X192 array. I have changed the array to 160X160 along with all code concerned with the array size.

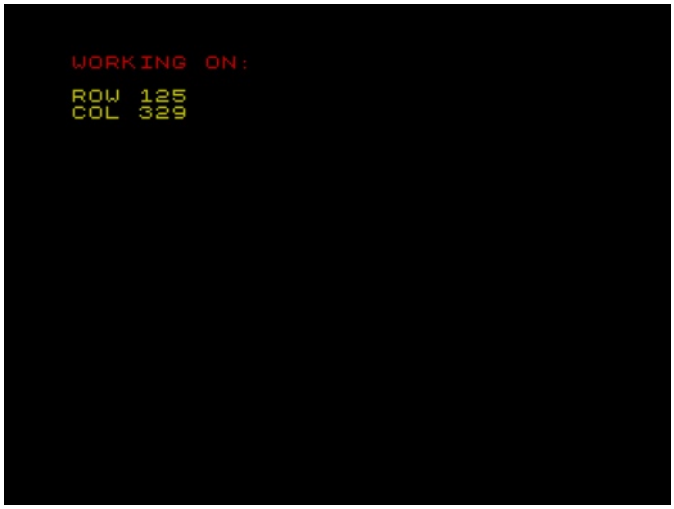
A plus from converting to the 2068 is that now you are not limited to the characters used by the ZX-81, but can design your own to make the printout picture look just as you wish. Most of you have your own UDG designer programs so I will not go into their creation here. In case you did not read the original article, when you have the array printed ten character/UDG's are used to create the 'gray scale' of the picture. Using different characters or UDG's has a great effect on the result and the artistic minded will enjoy the personalized touch which can be achieved this way.

Enter my listing and save it to tape before attempting to RUN it. This is always good practice and in the event of a crash will save loss of all of your work. UDG's you might want to use should be under letter A thru I inclusive. Your A UDG should be the lightest, that is the least black showing. The I should be completely black, or the darkest. The idea is to create a progressively darker character as you approach 'I'. These UDG's are then used to create the 'gray scale' when printing the hardcopy picture at the end of a run. One of these is selected based on the value of each location in the 160 X 160 array, lower values are the lightest, highest values darker.

One of my goals in doing this conversion was to be able to see the picture on the screen without the expense (and time) of printing it. Try the PREVIEW option and you will see a close resemblance to the printout with what appears to be elevation lines such as used in a topographical map. I liked the display with the lines as it gives the illusion of depth and visualizes the slopes in the data. Now you can get a good look at the array in a lot less time and possibly be encouraged to explore the Mandelbrot "world" for yourself.

My display is built by scanning the P$ array and using the array address to address a plot position on the screen. The picture is built starting at the lower left hand corner (which I call the origin), scanning across horizontally to the right. Subsequent lines build one above the other until the picture ends up with 160 dots across and 160 lines high.

Deciding whether to plot or not (that is the question!) is critical to how the finished picture will look. I experimented quite a bit to create something which would give you a good representation of what the printout would look like. Line 4030in my code decides to plot or not, the decision is based on three things. First the slope of the data is tested, if there is a change of 1 between the previous location (one location to the left) a dot will be plotted. The change can be either positive or negative, but must be only by 1. Next the same test is performed in the



vertical axis by comparing the data at the current address to that of the point directly below. This is done to produce a consistent display which is insensitive to the orientation of the shapes being represented. Lastly, if the data is 128, the highest

possible level, a dot will be plotted.

Try modifying the tests in line 4030 and see how different the result will be. I tried fancy schemes of testing for levels or looking for slopes greater than 1, but after all was said and done ended up with the code you see in the listing. For fun, try changing line 4030 to look for only rising data slope, "IF CODEP$(O,P)>CODE P$(O,P-1) THEN GOTO 4130". Experiment with testing for other things and see what results.

Taking advantage of the ability of the 2068 to save array files I added two options to the Mandelbrot program. Option 7 will save the array, all of the program variables (so you can save in mid run and continue later) and the date to help keep track of multiple files. Option 8 reloads everything and you can pick up where you left off in cases where the array computation was not complete. I did not save the "settings" for the printout thresholds, you may wish to expand on what I did to include them by enlarging the numeric array where the other variables are saved.

One feature I haven't mentioned is printout of the screen display generated by the PREVIEW option. Have your 2040 printer on and enabled and upon completion of the plotting a copy will be printed along with the coordinates you entered at the start of the run. The current loop count of variables M and N are also printed to let you know where things are in the event you do a PREVIEW before the array computation is completed. Sometimes you can see enough in a partially complete array to know whether or not to let it continue with the current coordinates. When you initially start a run, the coordinates entered will also be printed if the printer is on at that time. This makes a good reference to remind you what numbers you entered.

In exploring the plots I discovered that you can view the whole "world" in one picture. All other features exist within the "world" which you can see with these coordinates: A = -2.1,B = -2.1, S = 4.95. I have not seen any other features outside of this "world" and so far as I know this is all there is with the formulae used in this program.

Use the coordinates given in the previous paragraph and print the result. With a ruler draw two lines bisecting the outer circle on the printout. The lines represent the "zero" lines or coordinates for the A-CORNER (horizontal) and B-CORNER (vertical). This can be proved by computing the coordinates of A= 0, B = 0, s = 5. The result will have the lower left hand corner of the picture (the origin) at the intersection of the lines you drew on the first printout. Knowing this will give you a way to navigate around this Mandelbrot "world". Making the first or "A" coordinate positive moves the origin of the picture to the right of the vertical line, negative to the left. The "B"coordinate works the same way, but this time a positive number moves the origin above the horizontal line on your first picture, a negative number move it below.

With a little practice you can come very close to being where you want to explore in the Mandelbrot world. If you explore around the edges of the "black" areas you will see the canyon lands as I call them. With a little thought you can even draw a grid over the "world" which will help you navigate with some precision.

Once You get the origin of the picture at a interesting point you can magnify the features by using a smaller "S" number. The "S" number is the relative size of the result compared to the Mandelbrot "world", the smaller number gives you a smaller piece of it to fill the screen.

By the way, it is possible to speed this program up by compiling it with one of the commercially available compilers. I was successful using TIMACHINE, but this required some restructuring to make it work. One disappointment with the compiled code was that there was little increase in

the computation of the array. This code uses the floating point routines in ROM which cannot be speeded up.

OS-64

I thought I would tinker with Zebra's OS064 catridge (as a DOCK file) with Zesarux emulator. I have not used it so I wanted to see what it could do.

Loading the .dck file is easy enough with Zesarux, just go to the Storage menu, then Timex Cartridge, then Insert Cartridge, and then select the .dck file. The emulated system needs to be reset, and that is done via the Debug menu option. Once it is reset, the OS-64 cartridge kicks in and the system is changed to Mode 6, the 64 column mode. Despite it being 64 column mode, there is still only 22 rows on the screen, just like the normal 32 column mode.

The mode is monochrome, where only one INK and PAPER color is possible. If setting the PAPER color, the INK is set by the system. The same is true if setting the INK, the PAPER color is choosen. The border color is always the same color as the paper. The default PAPER with OS-64 is black with white INK.

Using Zesarux, the 64-column text is pretty clear. I think a good composite monitor is needed if using a real T/S 2068, as a TV does not have the resolution for this mode.

Mode 6 is really only for text. Graphics are available, but they are not hi-res and are the standard 256x192. FLASH does not work in this mode.

In normal mode, there is 38654 bytes of free memory. In 64 column mode there is 33727 bytes of free memory, a loss of 4927 bytes.

One of the main features of OS-64 is that it fixes a number of issues that the main ROM has with the 64 column node. With OS-64, BASIC defaults to 64-colum node, where LIST and PRINT statements work just fine.

BASIC programs will work, but they are using multiple INK or PAPER commands, then the screen will flicker as the whole screen changes for each INK statement. This might make those program difficult to use with OS-64.

There were a number of program released to utilize OS-64, but the only one that has been archived is the OS-64 Utilities #1 tape. Most of the programs on the tape are printer drivers. There is EDITOR64, a limited text editor written in BASIC. It essentially turns the whole screen into a text document that can be edited by moving the cursor around and adding or deleting text. The first time I ran EDITOR64 I has just listed the program. Instead of clearing the screen before the editor started, it started immediatly and I was able to "edit" the listing on the screen.

The editor does not seem to allow more than one page of text, nor do I see anyway to save the text to tape. I think this was more of a proof-of-concept or a demo than a full editor.

OS-64 does support user defined graphics (UDGs). I got an example program from the Spectrum manual that sets up a PI symbol as a UDG. When printed to the screen, it showed up. This means that limited games are possible with OS-64. I could imagine a monochrome version of Space Invaders being played.

I also tested some assembly programs. The first one I tested was a simple "print a string" that used the a ROM call to print the string. When it ran, it switched back to 32-column mode, back to 64-column mode and then basically crashed. I'm guessing there was something in the ROM call that was not compatible with 64 column node.

The second assembly program I tried used RST 16 to print characters to the screen and it worked. I'm guessing that the normal PRINT ROM routines do not work with OS-64. It is easy enough to not use the ROM routines, or it might be possible to see where they exist in the OS-64 ROM. Since the OS-64 ROM is just the original HOME ROM, modified and copied to the cartridge, some enterprizing soul could disassemble the OS-64 ROM and see what what locations are the new PRINT routines.